UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Una herramienta para la detección de plagios en código Ruby

Roberto García Teodoro
Tutora: Esther Guerra Sánchez

Mayo 2018

Una herramienta para la detección de plagios en código Ruby

AUTOR: Roberto García Teodoro
TUTORA: Esther Guerra Sánchez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2018

Resumen

La detección de copias en el ámbito académico resulta de elevada importancia, ya que las copias fraudulentas repercuten de manera muy negativa en el aprendizaje de los alumnos, además de las serias implicaciones morales que acarrean.

Por ello, el objetivo de este Trabajo de Fin de Grado es la creación de un sistema que permita detectar los plagios existentes en un conjunto de ficheros que estén escritos en código Ruby.

El profesor podrá utilizar el sistema para detectar las copias entre los ficheros que se encuentren en una ubicación local; posteriormente, podrá revisar los pares de ficheros que el sistema haya detectado como copias.

Proveer de un sistema que permita detectar dichas copias complementaría el sistema de detección de plagios con el que ya cuenta Moodle, lo que además llevaría a una reticencia por parte del alumnado a continuar realizando copias, pues el riesgo de ser detectado puede ser ahora mayor gracias al sistema.

La dificultad de proporcionar un sistema que detecte copias es elevada debido a múltiples factores, en primer lugar, se deben minimizar tanto el número de falsos positivos como el de falsos negativos, así como la dificultad técnica que acarrea detectar las copias, debido al hecho de que comúnmente, el estudiante intenta con los medios a su disposición hacer la copia lo menos parecido posible al original.

En cuanto al apartado técnico del trabajo, en primer lugar se analizará el estado del arte en cuanto a los métodos actuales de detección de copias y se explicarán brevemente cada una de las aproximaciones posibles para abordar el problema; después, se llevará a cabo el análisis para determinar los requisitos que deben ser satisfechos por la aplicación.

Más adelante se tratará el proceso de desarrollo de la aplicación, en la que se ha utilizado el lenguaje de programación Scala, haciendo uso de la librería JavaFX para el desarrollo de la interfaz de usuario.

En este trabajo se desarrollarán tanto el backend de la aplicación como una interfaz de usuario que permita la correcta interacción con el sistema.

Palabras clave

Ruby, Scala, AST, Detección de Copias, Plagio, JavaFX, Desarrollo Iterativo, Software Antiplagio.

Abstract

Copy detection in academia is of great importance, as fraudulent copies have a very negative impact on student learning, as well as moral issues that this implies.

Therefore, this Bachelor Thesis deals with the creation of a system that allows the detection of existing plagirarisms in a set of files written in Ruby.

The professor will be able to use this system to detect copies between files in his local filesystem; afterwards, he will be able to check those pairs of files which the system has detected as copies.

Providing a system that allows copy detection will complement the detection system already present in Moodle, and this would also lead to a reticent behaviour from students to keep copying, because the risk of being discovered can be higher now, thanks to this system.

Developing a system that detects copies is complex, due to multiple factors. First of all, both the number of false positives and false negatives must be minimized, and moreover, copy detection is technically difficult, because of the fact that usually, students do all in their power to create the most unequal copy from the original one.

Regarding the technical section of this thesis, the state of the art of the current methods of clone detection will be firstly analyzed, and each possible approach will be explained. Later on, in the analysis section of the document, we will list the requirements that should be satisfied to carry out the copy finding.

The application has been developed in Scala programming language, making use of the JavaFX library for the user interface development.

In this Bachelor Thesis, both the application backend and its frontend (allowing a correct user interaction) will be developed.

Keywords

Ruby, Scala, AST, Copy Detection, Plagiarism, JavaFX, Iterative Development, Plagiarism Checker Software.

Agradecimientos

A mi tutora, Esther Guerra, por su apoyo y su ayuda en la realización de este trabajo.

A mi familia y en especial a mi pareja, por su continuo apoyo y ánimo.

A mis compañeros de trabajo por responder a mis dudas cuando se me presentaba un reto técnico.

A mis compañeros de clase y profesores por contribuir a mi desarrollo intelectual y profesional estos años en la universidad.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

La duplicación de código es un problema que ha sido tratado múltiples veces en la ingeniería del software, incluso se ha llegado a principios en el desarrollo de software para evitar que las repeticiones de código ocurran, por ejemplo, el principio DRY (siglas de Don't Repeat Yourself) o la regla del número tres (si copias una determinada funcionalidad en un total de tres lugares, entonces debes sacar la funcionalidad a un único sitio).

Un caso particular de la repetición de código (al menos en lo que a su detección refiere) se trata de las copias de código fraudulentas, donde un individuo copia una sección de código de un origen ajeno. Si bien este caso tiene ciertas similitudes con la copia de un desarrollador o de un equipo de desarrollo de su propio código, también tiene ciertas diferencias, que serán analizadas e intentaremos evitar que se interpongan en la detección.

De manera más específica, el plagio en el entorno universitario es peligroso, ya que socava los principios de aprendizaje y de la honestidad entre los compañeros, además de las serias implicaciones morales.

En [15] se realizó una encuesta a estudiantes, de grado, posgrado y profesores contratados sobre las copias en trabajos escritos, con los siguientes resultados, expresados en porcentaje de alumnos que habían practicado el comportamiento descrito en el último año:

*Cheating on Written Assignments**

	Undergraduates	Grad Students	Faculty
Working with others on an assignment when asked for individual work	42%	26%	60%
Paraphrasing/copying few sentences from written source without footnoting it	38%	25%	80%
Paraphrasing/copying few sentences from Internet source without footnoting it	36%	24%	69%
Receiving unpermitted help from someone on an assignment	24%	13%	44%
Fabricating/falsifying a bibliography	14%	7%	34%
Turning in work copied from another	8%	4%	38%
Copying material almost word for word from a written source without citation	7%	4%	59%
Turning in work done by another	7%	3%	45%
Obtaining paper from term paper mill	3%	2%	29%

Figura 1-1 Encuesta sobre las copias en trabajos escritos [15] (tabla 4)

A la luz de estos resultados, se percibe la necesidad de detectar las copias realizadas en el entorno universitario, a fin de que éstas dejen de ser repetidas.

1.2 Objetivos

Por lo mencionado anteriormente, este trabajo tiene el objetivo de desarrollar un sistema usable que permita analizar un conjunto de ficheros escritos en Ruby para concluir si en esos ficheros se ha producido un plagio, y, en caso afirmativo, permitir comparar dichos ficheros.
Por tanto, se estudiará el estado del arte de la detección de copias de software, reconociendo los principales métodos y algoritmos para llevar a cabo dicha detección. Se implementará una selección de los algoritmos y se proporcionará una interfaz de usuario que permita la interacción con la aplicación.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 2 – Estado del arte:** En este capítulo se explica un poco de teoría sobre los métodos actuales para la detección de copias o repeticiones de código. Se tratan los diferentes métodos existentes a través de las aproximaciones que se tomen para resolver el problema, sean estas textuales, léxicas, sintácticas o semánticas.
- **Capítulo 3 – Análisis:** Sección en que se detallan los requisitos funcionales y no funcionales que tiene que satisfacer la aplicación.
- **Capítulo 4 – Diseño:** Apartado en el que se detalla la organización del proyecto, se explican los diferentes módulos y se proporciona un diagrama para visualizar su composición y relación.
- **Capítulo 5 – Desarrollo:** Se explica todo el proceso que llevó desarrollar el proyecto, desde las primeras pruebas de concepto hasta conseguir el resultado final.
- **Capítulo 6 – Pantallas:** Sección en que se muestran las diferentes secciones de la interfaz de usuario de la aplicación.
- **Capítulo 7 – Integración, pruebas y resultados:** Capítulo donde se detallan las pantallas resultado de la integración, así como los conjuntos de datos con los que se han realizado las pruebas para comprobar el correcto funcionamiento de la aplicación.
- **Capítulo 8 – Conclusiones y trabajo futuro:** En este apartado se detallan las conclusiones a las que se ha llegado con el desarrollo del proyecto, así como una serie de mejoras interesantes que se podrían realizar en dicho proyecto para que esté en la nube, se puedan obtener estadísticas complejas y más.

2 Estado del arte

2.1 Diferentes aproximaciones

Actualmente hay varias aproximaciones para resolver el problema de la detección de copias de código, las cuales procederemos a comentar a continuación. La mayoría de estas soluciones están enfocadas al problema extendido de la repetición de código por parte de un desarrollador o de un equipo de desarrolladores, que si bien tiene algo en común con la parte que a nosotros nos interesa en este trabajo (copias fraudulentas de código), no nos va a satisfacer plenamente, pues en el caso de las copias fraudulentas la persona que efectúa el plagio tiende, lógicamente, a ocultar la copia mediante el cambio de orden en las líneas, nombres, etc. La mayor parte de estas aproximaciones estarían pues pensadas para detectar copias que pueden suceder en el día a día de un desarrollador, como pueden ser el cortado y pegado de funciones para después hacer alguna pequeña modificación (o incluso ninguna modificación).

Como comentan C. K. Roy y J. R. Cordy [1], comparar las diferentes herramientas tiene grandes complicaciones debido a diferentes factores, entre los que destacan los siguientes:

- Diversidad de las técnicas para la detección.
- Falta de definición acerca de qué se considera una similitud o copia.
- Ausencia de benchmarks.
- Diversidad de lenguajes de programación presentes en los diferentes desarrollos.
- La alta sensibilidad de los parámetros con los que se debe proveer al programa para detectar copias.

Indagaremos más profundamente en este último punto en posteriores apartados de este trabajo.

Para este análisis del estado del arte vamos a basarnos en la completa investigación de [1], que determina que según el nivel de análisis aplicado al código fuente, podemos clasificar las diferentes alternativas de detección de copias en cuatro principales categorías, podemos encontrarnos con aproximaciones textuales, léxicas, sintácticas y semánticas.

Procedemos a detallar a continuación cada una de ellas:

2.1.1 Aproximaciones Textuales

Las aproximaciones textuales se basan en realizar comparaciones utilizando las propias líneas de código tal cual, sin normalizar ni aplicar transformaciones o unas transformaciones muy leves.

Podemos encontrar un ejemplo de esta aproximación en Ducasse et al. [2], donde transforman cada una de las líneas de código de los ficheros a comparar en lo que denominan *effective lines*, esto es, eliminando los comentarios y espacios en blanco. Después comparan todos los ficheros sospechosos de albergar fragmentos copiados y posteriormente extraen una matriz con los resultados de la comparación de ambos ficheros, donde un eje corresponde a cada una de las líneas de un fichero y el otro eje a las del otro fichero, en esa matriz se marca un punto cuando se detecta que dos líneas de ambos ficheros son iguales, de manera que cuando aparecen líneas diagonales estamos ante una secuencia de líneas de código copiadas.

Este proceso lo podemos ver en la figura 2-1

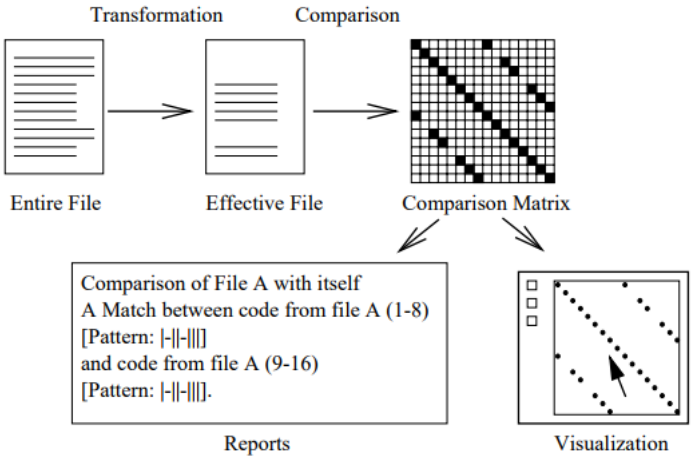

Figura 2-1 Representación de los pasos para la búsqueda de clones de código en Ducasse et al.

2.1.2 Aproximaciones Léxicas

Las aproximaciones léxicas comienzan transformando el código fuente en una secuencia de *tokens* léxicos, imitando el análisis léxico de un compilador. Después, sobre esa secuencia, se buscan subsecuencias iguales, que corresponden a los clones encontrados.
En esta aproximación podemos encontrar, por ejemplo, el programa CCFinder realizado por Kamiya et al. [3]. CCFinder se encarga de dividir las líneas del código en *tokens* utilizando un analizador léxico, después todos los *tokens* de los diferentes ficheros fuente se unen en una sola secuencia, se reemplazan identificadores y constantes por tokens genéricos.

2.1.3 Aproximaciones Sintácticas

Como su propio nombre indica, las aproximaciones sintácticas utilizan un analizador sintáctico para convertir el código en un árbol de sintaxis abstracta o AST, por sus siglas en inglés (Abstract Syntax Tree).
Los AST pueden ser comparados mediante la búsqueda de coincidencias entre los árboles generados o mediante el uso de métricas. Vamos a proceder a analizar ambos métodos:

- Aproximaciones sintácticas utilizando AST:
  Se caracterizan por primero generar un AST utilizando un analizador sintáctico específico para el lenguaje objetivo y después utilizan técnicas de comparación de árboles para encontrar los fragmentos de código copiados.
  Uno de los programas más destacados en esta categoría es *CloneDr*, donde, utilizando lo descrito en Baxter et al. [4], se procesa el código (en una fase muy parecida a un compilador) para generar un AST, a continuación se comparan los árboles generados utilizando una función hash. Entraremos a analizar este método en apartados posteriores de esta memoria, ya que tiene gran peso en la resolución de nuestro caso de uso concreto.

- Aproximaciones sintácticas utilizando aproximaciones basadas en métricas:
  En lugar de comparar el árbol entero recolectan métricas de los fragmentos de código para compararlas. En una primera instancia, se transforma el código a una AST o un Grafo de Control de Flujo (CFG por sus siglas en inglés, *Control Flow*

Graph). También se realizan hashes de clases, métodos, funciones… para después ser comparadas.
Los métodos más populares en esta categoría también se basan en comparaciones en bloques delimitados por un *begin* y un *end* (basándose mayormente en encontrar copias debidas a *corta y pega*) [5] y el entrenamiento de redes neuronales a través de fragmentos del propio código para detectar las copias [6].

2.1.4 Aproximaciones Semánticas

Por último, las aproximaciones semánticas utilizan un análisis estático del programa para dar más información que un análisis sintáctico. Encontramos dos tipos principalmente:

- Técnicas basadas en el Grafo de Dependencias del Programa (PDG por sus siglas en inglés, *Program Dependency Graph*):
  Llevan la abstracción un paso más allá, para ello calculan un PDG, para lo que utilizan información del flujo de control y el flujo de datos. Una vez generado el PDG se utiliza un algoritmo de búsqueda de isomorfismos para encontrar subgrafos semejantes.

- Técnicas híbridas:
  Combinan técnicas sintácticas (por el uso de métricas) con técnicas semánticas (por el uso de grafos).

2.2 Comparación de las diferentes aproximaciones

Las aproximaciones anteriores deben ser utilizadas según nuestras necesidades, así, por ejemplo, y tal como se comenta en [16]:

- **Aproximaciones textuales**: se trata de aproximaciones que, como hemos comentado anteriormente, no realizan análisis del código fuente antes de comparar, es por este motivo que destacan en su velocidad de procesamiento, así como en poder ser las más utilizadas para llevar a cabo comparaciones independientemente del lenguaje, ya que simplemente se basarán en la detección de cadenas de caracteres, a pesar de las dificultades para detectar dos fragmentos que hayan sido ligeramente modificados.
- **Aproximaciones léxicas**: algo más lentas en cuanto a algoritmia que las aproximaciones textuales, debido al proceso de transformación en *tokens*, detectará los clones que sean exactos, así como los que hayan sufrido ligeras modificaciones, pero tendrá más dificultades con los clones que hayan sufrido modificaciones profundas.
- **Aproximaciones sintácticas**:
  - **Usando AST**: requiere un procesamiento anterior mayor, debido a la transformación del código fuente en un AST, además luego se deberá implementar un algoritmo eficiente de comparación de árboles, es por eso que es una aproximación pesada en tiempo y uso de memoria, aunque su detección de duplicados es buena debido a su representación como árbol.
  - **Basadas en métricas**: su uso principal es la detección de copias de código a un nivel más alto, por ejemplo, una función repetida.
- **Aproximaciones semánticas**: a las técnicas basadas en PDG podríamos denominarlas como las aproximaciones más pesadas, ya que su proceso requiere,

como hemos comentado la generación de ese PDG, además, se debe comparar si ambos grafos son isomorfos, lo que constituye un problema NP.

Debido a este análisis, en este trabajo haremos un análisis de la aproximación textual, debido a su importancia como independiente del lenguaje, y posteriormente realizaremos una aproximación sintáctica con AST por su buen ratio de detección de clones respecto a su complejidad.

3 Análisis

3.1 Requisitos

En los siguientes apartados se definen el conjunto de requisitos que deben ser contemplados y resueltos por la aplicación a fin de satisfacer las necesidades de los usuarios.

3.1.1 Requisitos funcionales

- El sistema permitirá seleccionar los ficheros de una ubicación local.
- El sistema tratará exclusivamente ficheros escritos en el lenguaje de programación Ruby.
- El sistema reportará de una manera clara los ficheros que se han detectado como copia.
- Será posible visualizar cada par de ficheros acusados de copia en una misma ventana.
- El sistema permitirá el tratamiento de ficheros que hayan sido separados en diferentes carpetas por alumno o cualquier otro criterio de separación que el usuario haya escogido.
- Se dará un reporte del porcentaje de coincidencia por fichero, así como el porcentaje de coincidencia entre las carpetas.
- Se permitirá el cambio de los parámetros del algoritmo para poder hacer un uso avanzado de la aplicación.

3.1.2 Requisitos no funcionales

- El sistema funcionará en un tiempo inferior a 20 segundos para una práctica en la que se tenga un fichero por alumno.
- Se permitirá el uso de la aplicación a un único usuario de forma simultánea.
- El sistema usará menos de 1 GB de memoria RAM.
- Se dará una ayuda en la interfaz sobre el uso de cada parámetro.
- La interfaz de usuario será sencilla y usable.

4 Diseño

4.1 Organización del proyecto

El proyecto está dividido en tres sub-módulos, por un lado, tenemos el módulo que contiene la algoritmia de comparación de AST, por otro lado el módulo que contiene las funciones para la comparación de cadenas y por último, el módulo donde hemos implementado la interfaz de usuario.

- Módulo *string-comparison:* Ha sido desarrollado con el objetivo de explorar las diferentes vías para la detección de copias de código, aunque no tiene uso en la aplicación final.
    - Object RawStringComparator: compara cadenas que no han sido procesadas previamente.
    - Object PreProcessedRawStringComparator: tal como indica su nombre, compara cadenas cuyo texto original ha sido procesado previamente.
    - Trait Utilities: contiene las definiciones de funciones que son comunes a ambos ficheros, además de otras funciones de utilidad, como la obtención de los ficheros o la impresión de las matrices de resultados.

- Módulo *ast-comparison:*
    - Fichero Domain: contiene las definiciones de las clases que se utilizan en las funciones del resto de archivos:
        - Case class Position: clase que nos proporciona la información de la posición que ocupa un par de nodos en los ficheros originales. Está conformada por los nombres de ambos ficheros y la posición que ocupan cada uno de esos nodos en el correspondiente fichero.
        - Case class NodeComparison: clase que contiene la información para aplicar la fórmula de similitud que se propone en [4], a saber, nodos coincidentes en ambos árboles, nodos únicos en el árbol 1 y nodos únicos en el árbol 2, además contiene el factor de similitud, obtenido tras aplicar la fórmula descrita más adelante.
        - Case class Similarity: se trata de una clase cuyos dos parámetros son una instancia de Position y otra de NodeComparison.
    - Trait Utilities: al igual que en el módulo anterior, contiene funciones comunes a los siguientes ficheros, en este caso, tenemos funciones para el manejo de nodos, rangos de líneas, funciones hash, utilidades de comparación, etc.
    - Object RawAstComparison: se trata de la primera versión utilizando AST, contiene la funcionalidad necesaria para ejecutar una búsqueda de copias en las que se busca un número de nodos seguidos iguales mayor que un cierto valor definido, tal como se comentará en los siguientes apartados.
    - Object WeightedAstComparison: segunda de las versiones de detección de copias utilizando AST, aunque en esta versión se compara utilizando el concepto de similitud.
    - Object PaperComparison: contiene la última versión del algoritmo de comparación de árboles, tras analizar lo propuesto por Baxter et al. [4], es la versión que se utiliza finalmente en la aplicación.

- Módulo *ui*:
    - Class StartView: clase que representa la primera pantalla de la aplicación, en la que simplemente aparece un botón para dar paso a la siguiente vista, además de los logos de la Escuela Politécnica Superior y la Universidad Autónoma de Madrid.
    - Object MainView: Contiene los componentes necesarios para construir la pantalla principal de la aplicación, que permite la recepción de los parámetros necesarios para que se ejecute el algoritmo, además de recibir el reporte de los resultados.
    - Object ResultTab: Representa a las pestañas que se van a abrir para mostrar cada uno de los resultados positivos de la búsqueda de copias, en ella se muestran ambos ficheros sospechosos de ser copias a media pantalla, uno a la izquierda y otro a la derecha.
    - Fichero Utilities: Contiene funciones de utilidad, concretamente, para aplicar estilos, y funciones para validar los parámetros proporcionados por el usuario.

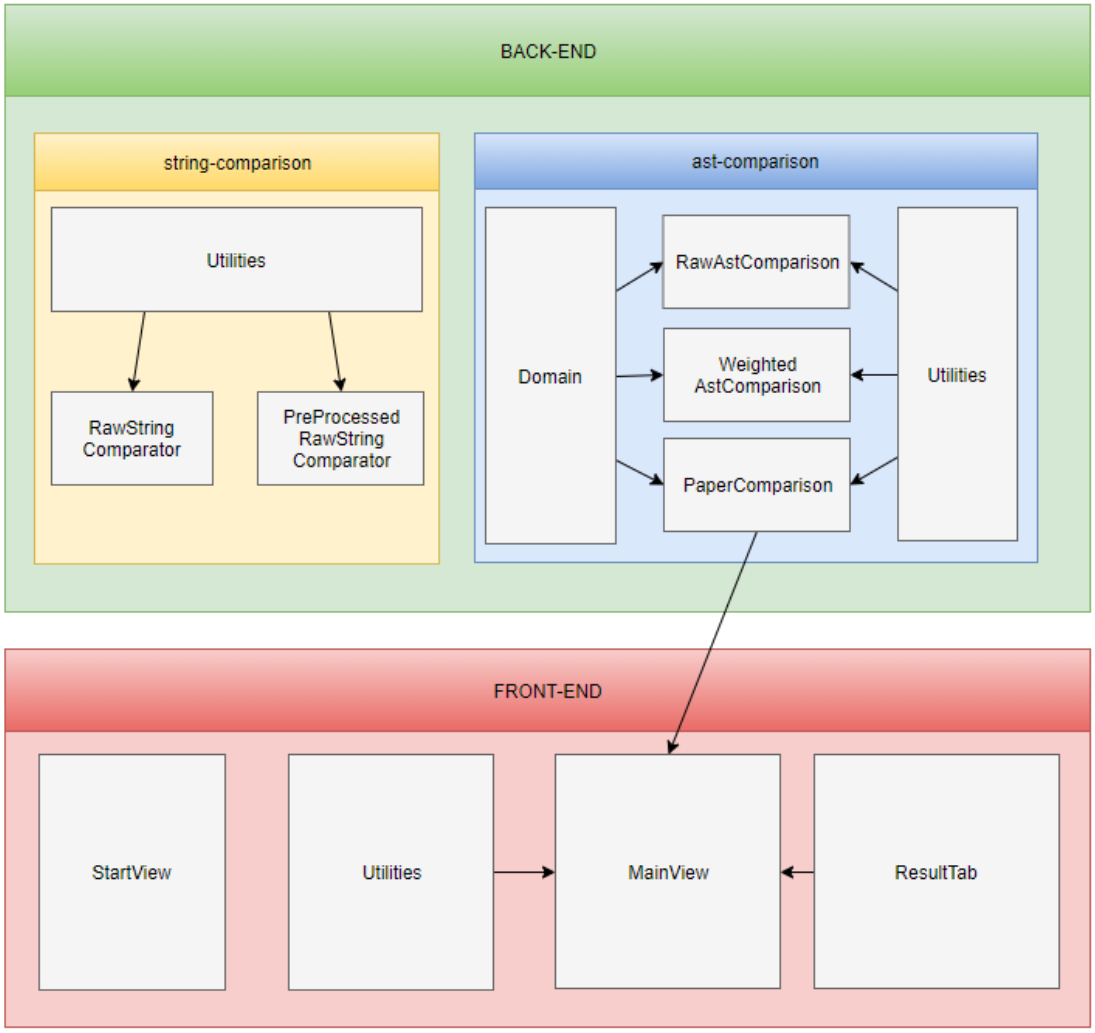

Figura 4-1 Diagrama de bloques representando la arquitectura de la aplicación.

5 Desarrollo

5.1 Elección de las tecnologías

5.1.1 Lenguaje de programación

Para realizar el proyecto utilizamos el lenguaje de programación **Scala** [7]. Elegimos Scala por tener mayor familiaridad con él, lo que nos permitía un mejor desarrollo, más rápido y ser menos propensos a cometer errores.

Scala es un lenguaje de programación creado en 2003 por Martin Odersky en la Escuela Politécnica Federal de Lausana (EPFL por sus siglas en francés, *École polytechnique fédérale de Lausanne*), mezcla el paradigma funcional con el orientado a objetos, permitiendo utilizar uno de ellos o mezclar ambos, por ello nos encontramos con características de los lenguajes orientados a objetos como el uso de clases, objetos, interfaces, herencia… junto a características más propias del paradigma funcional como podría ser la currificación, las funciones anónimas y de orden superior, *pattern matching*…

El sistema de tipos es estático, tratándose de un lenguaje fuertemente tipado, además Scala nos permite ser mucho más concisos en nuestro código, con lo que podemos hacer más funcionalidad escribiendo menos líneas.

Por otro lado, encontramos varias similitudes con el lenguaje de programación Java, hasta el punto de que muchas veces se asevera que Scala se trata de un "Java mejorado".

Las principales similitudes de Scala y Java podrían ser:

- Ambos funcionan sobre la JVM de Java, por lo que ambos se compilan al mismo bytecode, esto permite algo que es muy importante para el crecimiento de Scala: la enrome interoperabilidad con Java, ya que cualquier librería de Java puede ser utilizada libremente en Scala. Este punto es muy importante en la decisión de Scala como lenguaje de programación para este proyecto.
- Similitudes en el paradigma de la orientación a objetos.
- En ambos la sintaxis es muy parecida.

Hasta este punto podríamos pensar que Scala se trata de lo dicho anteriormente, simplemente un "Java mejorado", ahora bien, entre ambos podemos encontrar muchos elementos diferenciadores:

- Mientras Java tiene tipos primitivos (boolean, int…) en Scala todo son objetos, tal como los entenderíamos en Java. Las funciones también son objetos, esto nos permite pasar las funciones como parámetros a otros métodos.
- La convención de Scala nos indica que debemos seguir un patrón donde todos los objetos sean inmutables y no cambiemos las propiedades de ellos, algo que en Java no es común.
- Otra característica importante es que Scala permite la sobrecarga de métodos, donde los operadores o funciones tienen diferentes comportamientos según sus argumentos.

En conclusión, la mayor desenvoltura con Scala, así como su simplificación en su lectura y la posibilidad del uso de librerías de Java, hacen que nos decantemos por el uso de Scala en este proyecto.

5.1.2 Entorno de desarrollo

Una vez elegido Scala como lenguaje de programación, el siguiente paso fue buscar un entorno de desarrollo integrado (IDE, por sus siglas en inglés, *Integrated Development*

Environment) para comenzar a dar forma a nuestro programa. Por su calado en la comunidad Scala, decidimos utilizar el entorno IntelliJ IDEA [9], con un plugin para Scala, también es el IDE con el que nos sentimos más cómodos. El proyecto ha sido desarrollado principalmente en dos máquinas, una de ellas, un Intel i7 con 4 núcleos, 8 al tener hyperthreading, 16 GB de RAM, y sistema operativo Ubuntu 14.04 LTS, la otra tiene el mismo procesador, 8 GB de RAM y Ubuntu 16.04 LTS.

5.1.3 Control de versiones

Para el control de versiones se ha usado el software Git en el servicio de alojamiento Bitbucket [10]. Bitbucket nos ha permitido llevar un registro de cada uno de los cambios que íbamos incorporando al proyecto en forma de commits, así como permitirnos mantener el código subido para poder trabajar con él en su última versión utilizando Git.

5.2 Primera versión del desarrollo: aproximación usando comparación de cadenas (Aproximación Textual)

5.2.1 Análisis y prueba de concepto siguiendo la aproximación detallada en Ducasse et al. [2]

Para esta primera aproximación obteníamos cada uno de los ficheros que íbamos a comparar, para cada uno de ellos eliminábamos los espacios en blanco y posteriormente borrábamos todos los comentarios, tanto los de línea como los de bloque, para ello simplemente dejábamos todo el fichero como una sola cadena, en lugar de procesarlo línea a línea, por último, buscábamos todos los múltiples saltos de línea para dejarlos como uno solo.
Vamos a ilustrar esto con un ejemplo; En un cierto fichero, llamémoslo fichero completo, encontramos algo así:

```
=begin
Function that calculates Fibonacci secuence of a given number
=end
def fibonacci( n )


  return  n  if n <= 1
  #recursive
  ( fibonacci( n – 1 ) + fibonacci( n – 2 ) )
end
```

Llamaremos línea original a cada una de esas líneas.
Pero como hemos comentado anteriormente, con la aproximación dada en [2], queremos convertir estas líneas originales en lo que denominan “líneas efectivas de código”
Nuestro programa escribirá un fichero que contendrá lo siguiente:

```
def fibonacci( n )
  return  n  if n <= 1
  ( fibonacci( n - 1 ) + fibonacci( n - 2 ) )
end
```

Una vez que hemos convertido todos los ficheros completos en una colección ordenada de líneas de código efectivas, llegamos al siguiente paso, según se cuenta en [2], debemos comparar cada fichero consigo mismo línea por línea para detectar los fragmentos copiados, posteriormente formamos una matriz con las coincidencias o la ausencia de ellas y mostramos esa matriz, quedando como se expone a continuación:

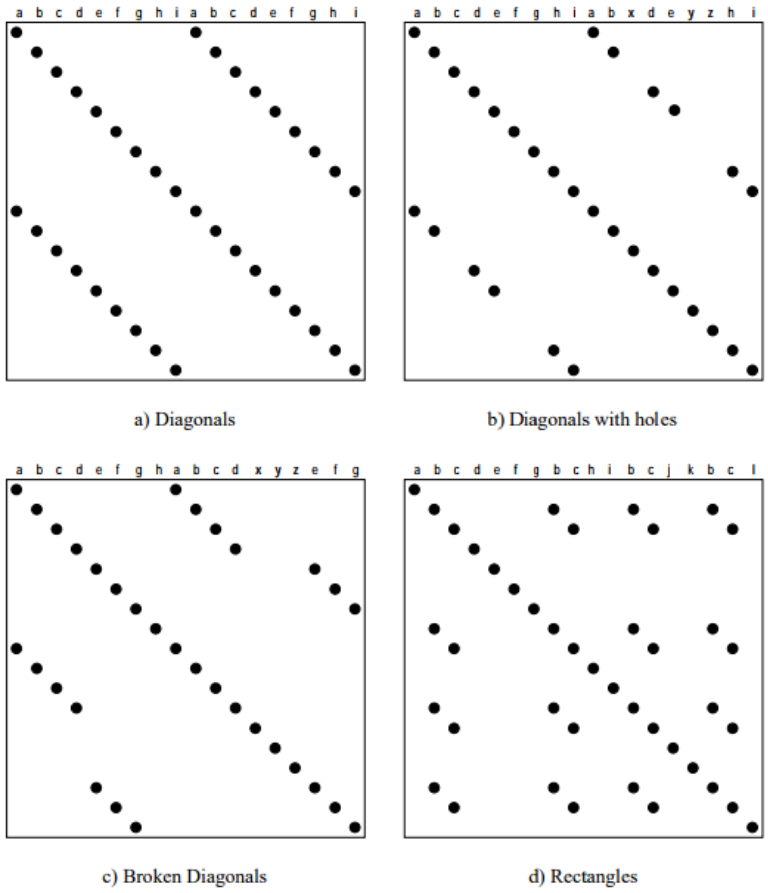

Figura 5-1 Representación de diferentes configuraciones de ficheros

Cada una de las matrices es el resultado de comparar cada línea del fichero con todo el fichero en sí. Por ello la primera fila de la matriz correspondería a la comparación de la línea *a* con cada una de las líneas del fichero (‘*a’* a ‘*g’*), la segunda fila a la comparación de *b* con todas las líneas y así sucesivamente.

Estos 4 ejemplos de matrices representados en [2], siendo su significado el siguiente:

a) Las diagonales indican secuencias de líneas copiadas.
b) Diagonales con huecos indican secuencias copiadas con algunas líneas cambiadas.
c) Diagonales desviadas indican que algunas líneas han sido añadidas en la secuencia copiada.
d) Configuraciones rectangulares apuntan a una repetición de las mismas líneas de forma periódica.

En nuestro caso, no nos interesa comparar cada fichero consigo mismo, pues estamos buscando copias entre las diferentes implementaciones de los alumnos por lo que de nada nos sirve el buscar en el mismo fichero, en nuestro caso concreto compararemos cada fichero con los demás.

5.2.2 Descarte de esta primera aproximación

Esta primera versión nos hace plantearnos algunas cuestiones.
¿Qué pasará con ficheros grandes? Entendiendo ficheros grandes como ficheros con más de cien líneas, nos quedarían matrices que apenas podríamos visualizar.
Damos con una segunda cuestión:
¿Un cambio en una variable o en el orden de las líneas podría hacer que no apreciáramos una copia?
El caso del cambio de nombre de una variable podría ser eludible, en una primera aproximación se nos ocurrió utilizar un nombre genérico para señalar las variables, métodos, campos de clases, etc., que nos permitiera ser agnósticos de los nombres. El cambio de orden de las líneas podría hacer que no visualizáramos correctamente las copias, pues alteraría el orden de la matriz y se podría dar el caso en que nos llevara a error, por lo que nos gustaría dar una respuesta más categórica, un "Sí" o "No", en lugar de dejar a la interpretación del usuario la detección de la copia por medio de la matriz. Para ser más claros, si va a ser tan tedioso revisar la matriz como comparar a ojo los pares de fichero, no estamos satisfaciendo los requerimientos de los usuarios. Además se presentan otros problemas que nos alejan más de esta solución, por ejemplo, esta opción nos gustaría más de cara a detectar copias de una misma persona o un mismo equipo, pero cuando intentamos detectar copias fraudulentas debemos cambiar el enfoque, ya que la persona que realiza la copia fraudulenta va a tratar de no ser detectada, eso va a hacer que utilice todas las características que el lenguaje pone a su disposición para evitarlo, es decir, utilizará una variable global en lugar de pasar un valor por parámetro o utilizará un bucle para sustituir alguna función de una biblioteca de listas, por ejemplo. En cualquier caso, podrían llegar a burlar a nuestro algoritmo. Es por ello que llegamos a la conclusión de que no queríamos comparar según lo que *estuviera escrito* en el código (aproximación textual), sino que queríamos comparar lo que el código *hace*. Por eso decidimos descartar esta opción y comenzamos a buscar una aproximación utilizando árboles de sintaxis abstracta.

5.3 Segunda versión: primera aproximación a los AST (Aproximación Sintáctica)

5.3.1 Selección de una librería para la generación de los AST

Habíamos decidido trabajar utilizando AST, esto implicaba cambiar el foco del problema, pasar de "¿qué hay escrito?" a "¿qué quiere decir/hacer lo que hay escrito?" y como hemos comentado, el primer paso consistía en transformar el código ruby de los ficheros a analizar en árboles de sintaxis abstracta, para ello buscamos una librería que cubriera esa funcionalidad. La primera que encontramos fue *GumTree* [11], se trata de un framework que nos permite transformar un fichero fuente escrito en alguno de los lenguajes soportados (Java, C, Javascript y Ruby) en un AST independiente del lenguaje, nos permite exportar el AST generado en varios formatos, computar diferencias entre dos AST, exportar las diferencias y visualizarlas gráficamente.

Hicimos una primera prueba de concepto utilizando el API que nos proporciona GumTree, obviamos la funcionalidad de la comparación de los árboles pues solo nos interesa la transformación de los ficheros ruby en AST.

5.3.2 Desarrollo de la primera prueba de concepto

Ya teníamos una librería para convertir nuestros ficheros ruby en árboles de sintaxis abstracta, así que el siguiente paso consistía en hacer una primera prueba de concepto o POC utilizando la librería para comprobar que efectivamente era lo que necesitábamos.
Comenzamos haciendo un programa simple:

```
val input: String =
  """def foo(bar)
    | puts bar
    | bar = 1
    |end""".stripMargin

val tree: TreeContext = new RubyTreeGenerator().generateFromString(input)
println(tree.toString)
```

Explicado brevemente, tenemos una cadena que representa un posible contenido de un fichero ruby (input), posteriormente usando el generador de AST para código ruby que nos proporciona GumTree, generamos un árbol utilizando la cadena input, para después imprimir por pantalla el árbol en cuestión. Obtenemos el siguiente resultado:

```
(() (102 "ROOTNODE" "" ((0 38)) (
   (60 "NEWLINENODE" "" ((0 38)) (
      (28 "DEFNNODE" "" ((0 38)) (
         (112 "METHODNAMENODE" "" ((4 3)) ()
         (4 "ARGSNODE" "" ((7 5)) (
            (6 "ARRAYNODE" "" ((8 3)) (
               (5 "ARGUMENTNODE" "" ((8 3)) ())))
         (13 "BLOCKNODE" "" ((15 20)) (
            (60 "NEWLINENODE" "" ((15 11)) (
               (39 "FCALLNODE" "" ((15 9)) (
                  (6 "ARRAYNODE" "" ((20 4)) (
                     (54 "LOCALVARNODE" "" ((20 4)) ())))
            (60 "NEWLINENODE" "" ((26 9)) (
               (53 "LOCALASGNNODE" "" ((26 8)) (
                  (40 "FIXNUMNODE" "" ((32 2)) ()))))))))
```

Podemos percibir una correlación entre los nombres de los nodos del árbol y nuestro pequeño código de prueba.
Tenemos un inicio (rootnode), y una línea inicial (newlinenode) en la que tenemos un def (defnnode), con un nombre de método, unos argumentos y un bloque de código (los tres están al mismo nivel en el árbol), etc.
Dado que trabajar con esta visualización de los árboles se hacía complicado y a que la funcionalidad de librería GumTree era mucho más amplia de lo que necesitábamos, tomamos la decisión de entrar en el repositorio de Github de GumTree [11] y aprovechándonos de su condición de software libre ver qué librería usaban para convertir el código fuente en AST.

Cuando entramos a observar el código de la clase *RubyTreeGenerator.java* encontramos lo que estábamos buscando al inicio del fichero:

```
import org.jrubyparser.CompatVersion;
import org.jrubyparser.Parser;
import org.jrubyparser.ast.*;
import org.jrubyparser.parser.ParserConfiguration;
```

Una vez vimos que utilizaban la librería jruby-parser [12] decidimos desarrollar una segunda prueba de concepto con ella.

5.3.3 Desarrollo de una segunda prueba de concepto

Comenzamos la nueva prueba de concepto utilizando jruby-parser, se trata de una librería para convertir ficheros ruby en AST, usada por NetBeans y Eclipse en sus módulos para ruby.
En esta primera versión, procedíamos a transformar el fichero haciendo uso del método *parse* de la clase Parser, ese método nos devuelve una referencia al *Node* raíz del árbol.
Para la primera versión lo que intentamos fue encontrar si dos AST tenían un camino de nodos iguales de una cierta profundidad. Esto se pudo hacer de manera sencilla mediante la definición de un par de funciones recursivas.

En primer lugar la función *check*, recibe dos nodos que la primera vez serán los dos nodos raíz mientras que las veces sucesivas node1 serán los hijos del node1 original y node2 siempre se mantendrá constante, representando un árbol entero.

```
def check(node1: Node, node2: Node): Int = {

  if (node1.childNodes().asScala.nonEmpty) {
    List(
      oneWithAll(node1, node1, node2),
      node1
        .childNodes()
        .asScala
        .map { node1Child ⇒
          check(node1Child, node2)
        }
        .max
    ).max
  } else {
    oneWithAll(node1, node1, node2)
  }
}
```

Figura 5-2 Función *check*

Como podemos ver esta función crea una lista cuyo primer elemento es una llamada a *oneWithAll* y el siguiente una llamada recursiva a *check* en la que se pasa como node1 a cada uno de los nodos hijos de node1.

Por otro lado tenemos la función *oneWithAll*:

```scala
def oneWithAll(originalNode: Node,
               node1: Node,
               node2: Node,
               inARow: Int = 0): Int = {

  if (node2.childNodes().asScala.isEmpty) {
    if (node1.getNodeType == node2.getNodeType) {
      inARow + 1
    } else {
      inARow
    }
  } else {
    node2
      .childNodes()
      .asScala
      .map { node2Child ⇒
        if (node1.getNodeType == node2.getNodeType) {
          if (node1.childNodes().isEmpty) {
            0
          } else {
            node1
              .childNodes()
              .asScala
              .map { node1Child ⇒
                oneWithAll(originalNode, node1Child, node2Child, inARow + 1)
              }
              .max
          }
        } else {
          oneWithAll(originalNode, originalNode, node2Child)
        }
      }
      .max
  }
}
```

Figura 5-3 Función *oneWithAll*

En este caso, tenemos una función recursiva que cada vez que encuentra que en ambos árboles hay un nodo igual, va al hijo de ese nodo y comprueba si en ambos árboles también es igual, y así sucesivamente.

Como podemos advertir de manera sencilla, este método puede tener importantes deficiencias, por ejemplo, cambios en el código pueden seguir afectando a nuestra capacidad para decidir si dos ficheros son copias o no, ya que, por ejemplo, tenemos un cierto nodo para indicar que estamos en el contexto de una llamada a un método o función. Pero a la hora de codificar podríamos llamar a una función o trasladar el código de esa función al flujo principal, al meter ese nodo de llamada a función en nuestro árbol haría que ya ambos árboles no pasaran nuestro filtro de ir encontrando varios seguidos, pese a que ambos fragmentos de código hagan la misma funcionalidad.

5.4 Tercera versión: versión definitiva usando AST

5.4.1 Mejorar el método de buscar caminos ininterrumpidos

Comenzamos a buscar referencias sobre cómo otros equipos se enfrentaban a retos similares a la hora de trabajar con AST. Volvimos a encontrarnos con el estudio de Baxter et al. [4], sobre el que habíamos leído previamente en nuestra fase de investigación de este mismo proyecto. Fue ahí donde encontramos lo que ellos denominan similitud, que podríamos definir como el factor de parecido entre dos árboles. Ya que como comentan en el estudio: "En lugar de comparar árboles por igualdad exacta, comparamos por similitud, usando unos pocos parámetros. El parámetro de umbral de similitud permite al usuario especificar cuánto similares deberían ser dos subárboles"

En Baxter et al. definen la similitud de la siguiente manera:

$$\text{Similitud} = 2 \times S / (2 \times S + L + R)$$

Donde S es el número de nodos compartidos por ambos árboles, L el número de nodos diferentes en el subárbol 1 y R el número de nodos diferentes en el subárbol 2.

De modo que procedemos a utilizar ese nuevo conocimiento para mejorar nuestro anterior algoritmo, en este caso nuestra antigua función *oneWithAll* ya no contará el número de nodos seguidos, sino que llamará a una función que nos comparará los árboles y nos devolverá un decimal entre cero y uno, luego deberemos filtrar los resultados que sean mayores que un cierto número, ese será nuestro umbral de similitud, el filtro que nos permita decidir a qué llamamos que dos árboles sean iguales.

Una vez que tenemos codificada esta vía, procedemos a probarla, lo primero que vemos es que necesitamos ampliar la memoria con la que dotamos a la JVM, pues un par de veces nos encontramos con errores por falta de memoria.
También notamos que el tiempo que tardamos en dar un reporte de los resultados es elevado, del orden de 5 minutos.
Esos dos problemas nos hacen plantearnos la necesidad de aligerar tanto la carga en memoria como los procesos para devolver un resultado efectivo. Así que recuperamos de nuevo el estudio de Baxter et al. con el objetivo de encontrar cómo han lidiado con algunos problemas parecidos.

5.4.2 Solución a los nuevos problemas y versión definitiva

Recopilando información en el artículo de Baxter et al. encontramos que detallan que las comparaciones tal como las hacemos tienen un orden de complejidad de $O(N^4)$, nos reafirma en la necesidad de mejorarlo y nos indica que vamos por el buen camino leyendo su trabajo, pues se han encontrado con problemas como los que nos ocurren ahora.
En el artículo relatan su manera de reducir las comparaciones de los árboles mediante el uso de funciones hash, para después pasar a comparar únicamente los árboles que tengan un mismo hash. A diferencia de lo que se podría pensar, esas funciones hash no pueden ser "perfectas", ya que entonces solo estaríamos comparando árboles que fueran exactamente iguales. Es por tanto necesario definir una función hash propia que sea intencionalmente mala, entendida mala como la negación de la perfección mencionada anteriormente.
Dado que en el artículo les da buen resultado una función hash que ignore los subárboles pequeños, de modo que procedemos a hacer algo de la misma forma y hacemos una

función hash que se calcule a partir de un árbol sin contar sus hojas. Esa función se hará para cada uno de los subárboles de cada árbol, minimizando así la posibilidad de que se escapen copias por inserción de nodos en el árbol, como comentamos anteriormente.
Ahora comparamos todos los árboles que hayan caído en el mismo valor del hash, lo que reduce de forma muy considerable la memoria y mejora muy notablemente el desempeño del algoritmo que en este punto tan solo tarda unos segundos en realizar su tarea.

5.5 Interfaz de Usuario

5.5.1 Primera versión usando ScalaFx

Para implementar la interfaz de usuario primero decidimos hacer uso de ScalaFX [13] al tratarse de una envoltura de los métodos de Java utilizando Scala para hacer que sean más idiomáticos para trabajar con ellos desde un proyecto Scala.
Al hacerlo así nos encontramos un problema, la falta de documentación y ejemplos de ScalaFX hacía que con nuestro poco conocimiento del framework no pudiéramos trabajar con soltura con él, eso nos hizo decantarnos por utilizar JavaFX, del que sí podíamos encontrar una amplia variedad de documentación y ejemplos prácticos.

5.5.2 Segunda versión usando JavaFx

Al trabajar con JavaFx pudimos nutrirnos de tutoriales [14] que nos ayudaron a comenzar a trabajar con el framework ya que no habíamos trabajado con él con anterioridad.
Al ser algo en lo que no teníamos soltura nos ha costado llegar al resultado final, ya que es algo muy específico. Como hemos comentado anteriormente, los tutoriales nos han ayudado bastante a conseguirlo.
La interfaz consta básicamente de una pantalla principal que presenta la aplicación. Contiene un botón que nos permite avanzar a la vista principal.
La vista principal contiene unos campos para introducir los parámetros y lanzar la ejecución, esta pantalla también nos presentará los resultados con el porcentaje de afinidad entre ficheros y entre carpetas.
Por cada par de ficheros detectado como copias se abrirá una pestaña con la pantalla dividida en dos en la que podremos repasar los dos ficheros que se han detectado como copias.
Las partes más complicadas de este apartado fueron entre otras:

- Lidiar con el scroll de las pantallas: esto fue complicado puesto que se detectó la necesidad avanzado el desarrollo y tuvimos que rehacer una sección importante de nuestra interfaz.
- Errores al tratar con el nombre de las carpetas según el uso fuera en Linux o Windows.
- La composición de la escena que utiliza JavaFX, en el que tenemos que definir paneles y tratar qué paneles están contenidos en otros, etc.

6 Pantallas

Tras iniciar la aplicación, llegamos a la pantalla de bienvenida, donde se muestran los logos de la Escuela Politécnica Superior y de la Universidad Autónoma de Madrid.

Figura 6-1 Pantalla de bienvenida de la aplicación

Tras pulsar el botón "Comenzar aplicación" avanzamos a la siguiente pantalla, donde podemos introducir los parámetros necesarios para la ejecución del algoritmo:

- La ruta donde se encuentra la carpeta que contiene los ficheros que van a ser analizados.
- El umbral de masa: el valor seleccionado como umbral de masa hace referencia al tamaño mínimo que deben tener los subárboles para ser tenidos en cuenta. Un valor alto implica que se buscarán fragmentos de código mayores, mientras que uno más bajo permitirá detectar copias localizadas en fragmentos menores. Tras los resultados de las pruebas realizadas, se recomienda un valor de, aproximadamente 40.
- El factor de comparación: El factor de comparación debe ser un número mayor que 0 y menor que 1, indica si se descartan o no los subárboles según el siguiente criterio: si el factor de comparación introducido es mayor que el calculado por la fórmula de similitud del algoritmo, el subárbol se descartará. Se recomienda proporcionar un valor ≥ a 0.85

Todos ellos tienen un botón con una interrogación al lado, en el que se explica su significado.

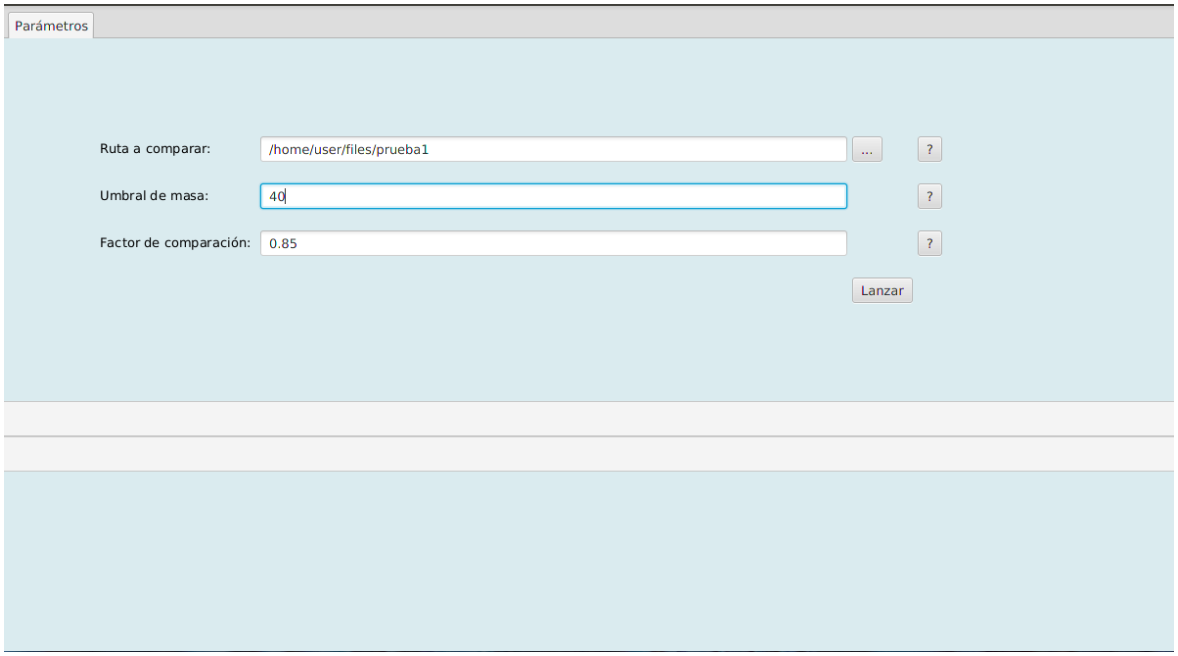

Figura 6-2 Pantalla principal de la aplicación, donde se permite introducir los parámetros para la ejecución del algoritmo

Tras introducir los parámetros apropiados, podemos pulsar el botón “Lanzar”, al hacerlo, se ejecutará el algoritmo, cuando haya terminado, y si procede, se nos mostrarán los resultados correspondientes de tres maneras:

- Bajo los parámetros, aparece un reporte de los ficheros en los que se han encontrado copias, mostrando el porcentaje que tienen en común. Los resultados están numerados coincidiendo con el número mostrado en la pestaña que compara ambos ficheros
- Bajo ese reporte, aparece las coincidencias agregadas por carpeta, mostrando también el porcentaje de parecido entre ambas.
- Por último, tenemos cada una de las pestañas que representa una pareja de ficheros que han dado positivo en el algoritmo. Al pulsar sobre cada una de ellas, se nos llevará a la pantalla de resultado, donde podemos comparar ambos ficheros.

En las siguientes imágenes podemos apreciar esto, en la Figura 6-3 vemos los reportes de coincidencia de ficheros y carpetas, además de las pestañas en la parte superior.
En la figura 6-4 vemos el reporte resultado del algoritmo, donde vemos los dos ficheros que se han detectado como copia.

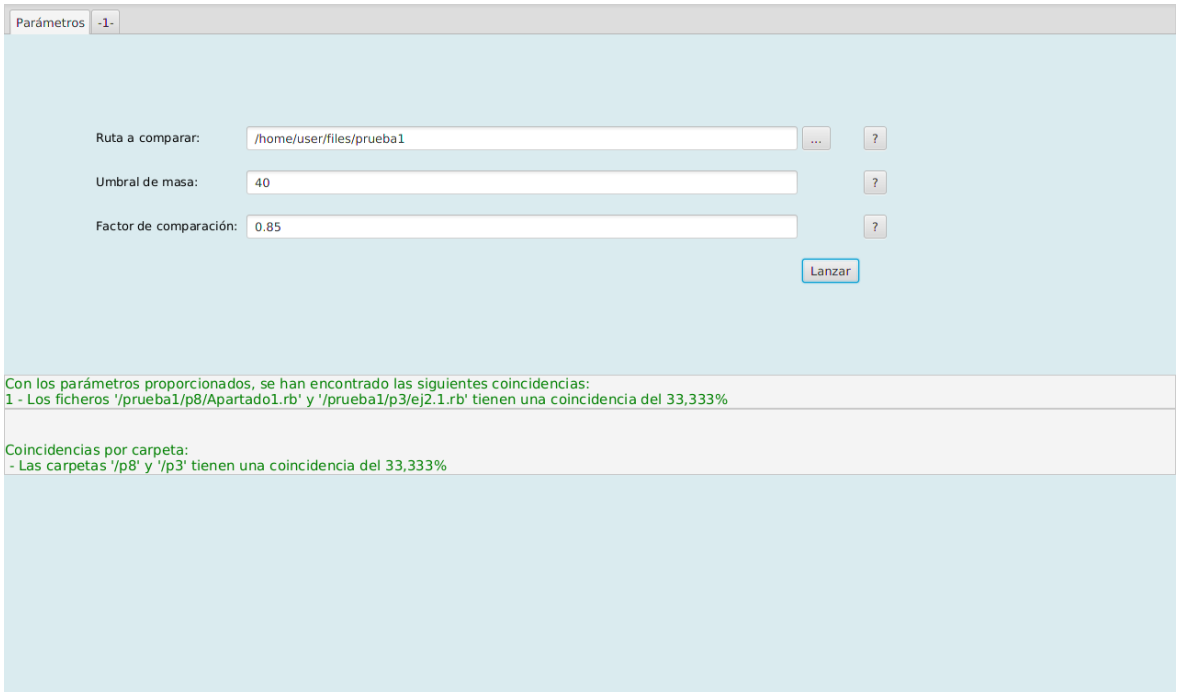

Figura 6-3 Pantalla principal mostrando el reporte de los resultados tras ser ejecutado el algoritmo

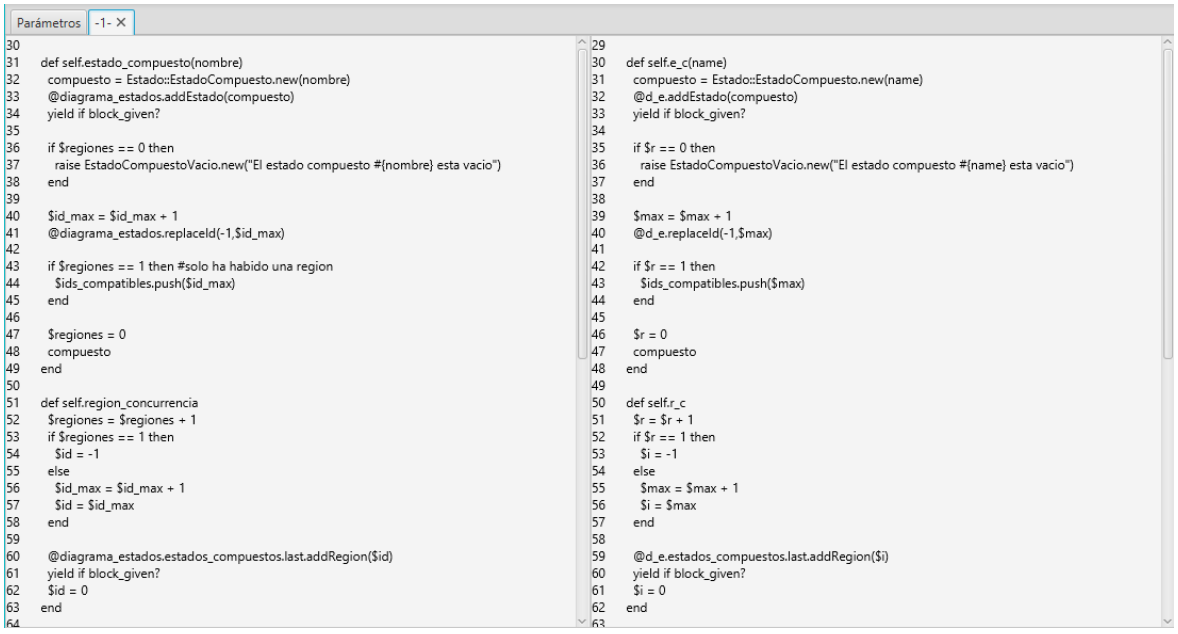

Figura 6-4 Pantalla de resultado que permite comparar dos ficheros en los que se ha detectado plagio

Por último, en la Figura 6-5, vemos cómo queda la pantalla cuando el reporte de errores es mayor de un único fichero. Como podemos ver en este caso, hay algunas prácticas de estudiantes que tienen coincidencias con otros ficheros de ellas mismas.

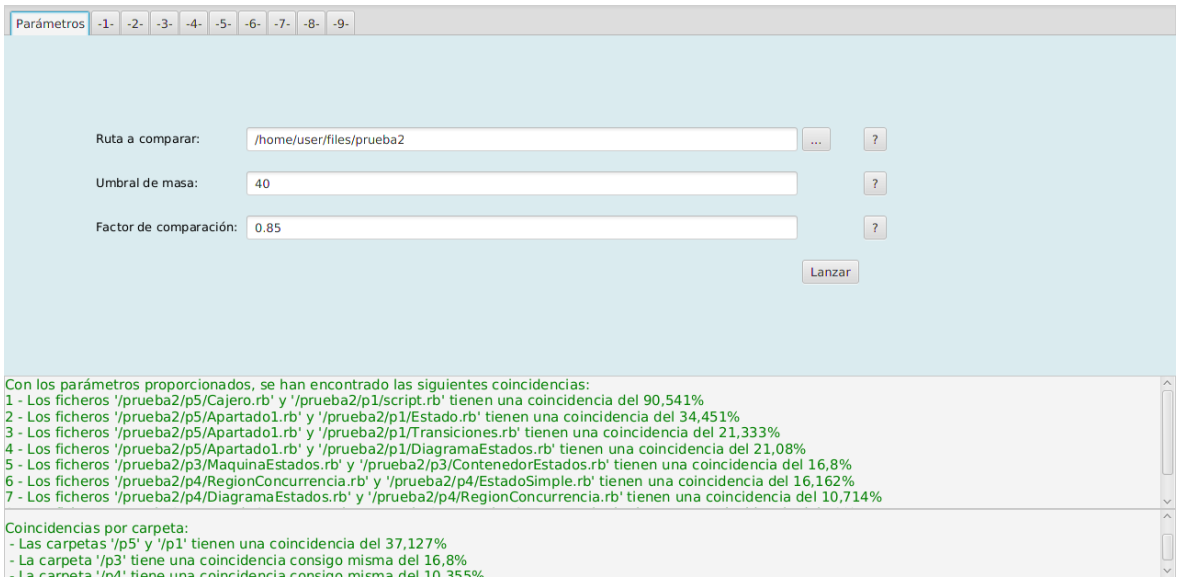

Figura 6-5 Pantalla principal mostrando el resultado de un set de pruebas mayor

Que coincidan con ellas mismas tiene una explicación que detallamos a continuación: Efectivamente el algoritmo ha acertado, estos dos ficheros tienen una porción de código que es idéntica, esto es debido en este caso a que el estudiante no ha modularizado una función y la ha duplicado en varios ficheros, por lo que, siendo cierto que los ficheros contienen copias, en este caso para nosotros sería un falso positivo.

7 Integración, pruebas y resultados

Con esta versión de nuestro programa, somos capaces de detectar satisfactoriamente los ficheros que contienen copias en todos nuestros sets de pruebas.
Se han llevado a cabo varias pruebas utilizando como set de datos prácticas reales realizadas por el alumnado de la Escuela Politécnica Superior, en concreto se trata de dos prácticas, estas prácticas son anónimas, por lo que no incluyen el nombre de ningún estudiante y han sido utilizadas con el único objetivo de probar el resultado de este trabajo.
La primera de ellas consta de diez carpetas, presumiblemente una por alumno o pareja de alumnos, cada una de ellas consta típicamente de un solo fichero (en función de cómo haya modularizado el código el alumno)
La segunda de ellas está formada por siete carpetas, que contienen entre cuatro y seis ficheros de código cada una.
El tiempo que toma el algoritmo en realizar las comprobaciones necesarias es del orden de unos pocos segundos, y el usuario tiene una experiencia de inmediatez en el flujo de trabajo con la aplicación.

Práctica	**Umbral de Masa**	**Factor de comparación**	**Resultados encontrados**	**Falsos negativos**	**Falsos positivos**
1	20	0.80	43	-	41
		0.85	42	-	40
		0.90	38	-	36
		0.95	35	-	33
	30	0.80	4	-	2
		0.85	4	-	2
		0.90	3	-	1
		0.95	1	1	-
	40	0.80	2	1	1
		0.85	1	1	-
		0.90	1	1	-
		0.95	0	-	-
2	20	0.80	26	-	24
		0.85	17	-	15
		0.90	9	-	7
		0.95	7	-	5
	30	0.80	15	-	13
		0.85	11	-	9
		0.90	8	-	6
		0.95	7	-	5
	40	0.80	14	-	12
		0.85	9	-	7
		0.90	6	-	4
		0.95	6	-	4

Tabla 7-1 Resultados del algoritmo según práctica, umbral de masa y factor de comparación

Tal como podemos ver en la tabla anterior, a medida que vamos aumentando el factor de comparación, disminuye el número de resultados encontrados, como también ocurre al subir el valor del umbral de masa.
En la tabla hemos contado como falsos positivos todos aquellos positivos que no corresponden exactamente con prácticas de diferentes alumnos y cuya copia sea sustancial, es decir, estamos considerando como falsos positivos el caso en que un estudiante repite una función en su propio código (lo que sí es en realidad una copia, pues se trata de una sección de código copiada y pegada) como en el caso del constructor de una clase, que al tratarse de la misma práctica, es lógico que sea parecido en los ficheros de los diferentes alumnos.
En los casos en los que hemos detectado falsos negativos, la causa ha sido que la parte de copia era pequeña, por lo que a valores grandes de umbral de masa/factor de comparación se perdían.
Dar un porcentaje de acierto siendo categóricos ante estos resultados es algo arriesgado, ya que siendo exactos, como hemos comentado anteriormente, la mayor parte de los falsos positivos no son tales en valores altos de umbral de masa.
En cualquier caso, se hace necesaria una labor humana de análisis e interpretación de los resultados, que requerirá la detección de los valores adecuados de umbral de masa y factor de comparación a fin de detectar adecuadamente las copias, así como un trabajo posterior de revisión de los resultados utilizando la herramienta de comparación de ficheros de la que dispone la aplicación.
En las pruebas realizadas el sistema mantuvo el consumo de memoria por debajo de los 500 MB.

8 Conclusiones y trabajo futuro

8.1 Conclusiones

En conclusión, este trabajo nos ha permitido analizar los métodos actuales de detección de clones de código, hemos aprendido sobre los diferentes problemas que acarrea la copia de código, no solo fraudulenta, sino los diferentes problemas que puede ocasionar cuando la repetición del código se produce en un entorno de desarrollo profesional, ocasionando grandes pérdidas de tiempo y de calidad del producto desarrollado, incrementando costes de mantenimiento.

Hemos aprendido también sobre el flujo de trabajo en un proyecto, donde hemos comenzado con una necesidad, en este caso, detectar copias fraudulentas de código entre los alumnos, hemos investigado herramientas similares, encontrando una dificultad, que en los entornos profesionales (para los cuales estaban pensadas las soluciones que analizamos) tenía otro enfoque, ya que les interesa detectar copias de código ocurridas en un mismo proyecto o incluso dentro de un mismo fichero. Eso difería de nuestro caso de uso, donde teníamos otras ciertas dificultades, ya que las copias debían ser detectadas en prácticas de alumnos, lo que planteaba dos problemas principales, por un lado, el alumno que practica la copia intenta con los medios de los que dispone que no sea detectada por el docente o por el programa que detectará la copia, el otro de los problemas es que para todos los alumnos, la funcionalidad es la misma, es decir, es normal que sus prácticas tengan cierto parecido, puesto que todos están desarrollando la misma funcionalidad.

Una vez analizadas las alternativas del estado del arte y comprendido plenamente el problema al que nos enfrentábamos, hemos procedido a un desarrollo iterativo guiado mediante pruebas de concepto, donde hemos analizado varias de las soluciones que se han aplicado en otros sistemas, hemos detectado los puntos en los que no nos encajaban, buscamos una nueva solución y repetimos la prueba de concepto. Este proceso nos ha llevado al uso de los AST, cuando nos encontramos con problemas con ellos procedimos a estudiar cómo trabajaban con ellos en otros equipos con problemas parecidos, mejorando finalmente nuestro trabajo.

Por otro lado, fuimos capaces de construir una interfaz que se ajustara a lo que los posibles usuarios necesitarían de nuestra aplicación, permitiéndoles tanto el ajuste de los parámetros necesarios para el proceso como una visualización adecuada de los resultados de la detección.

8.2 Trabajo futuro

Tenemos bastantes ideas para un trabajo futuro, en primer lugar, deberíamos intentar disminuir aún más tanto el consumo de RAM como el de CPU, con lo que podríamos reducir el tamaño a un microservicio, este microservicio podría ser desplegado en un entorno en la nube, como por ejemplo Amazon Web Services, lo que nos permitiría levantar varias instancias del microservicio, para soportar múltiples peticiones concurrentes.

Cambiaríamos la interfaz realizada en JavaFX por una interfaz en Javascript que funcione sobre el navegador, lo que permitiría que el usuario accediera con una identificación a un sistema remoto, haciendo log in en nuestra aplicación sin que tuviera que tener nada en su máquina local.

Por otro lado, cuando el usuario seleccionara los archivos que analizar, éstos se subirían a un sistema de almacenamiento remoto como S3, lo que permitiría en un futuro seleccionar los conjuntos de datos sin que el usuario dispusiera de ellos en su ordenador.
También se podrían llevar a cabo trabajos de análisis de los datos por docente, estudiante, práctica concreta, etc. utilizando los archivos subidos anteriormente, esto se podría realizar utilizando Apache Spark.
Incluso sería posible imitar el sistema de detección que tiene Moodle actualmente, ampliándolo al análisis de código, dado que Moodle solo realiza detecciones de copias de archivos de texto.
Por otro lado, el microservicio podría utilizar patrones de diseño reactivos, de forma que, por ejemplo, se permitiera el trabajo en streaming desde la subida del fichero hasta el reporte de los resultados, reduciendo así el uso de recursos.

Referencias

[1] C. K. Roy, J. R. Cordy, “Scenario-based comparison of clone detection techniques.”, Program Comprehension, 2008. 16th IEEE International Conference. IEEE, 2008. pp. 153-162.

[2] S. Ducasse, M. Rieger, S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code”, ICSM, 1999, pp. 109-118.

[3] T. Kamiya, S. Kusumoto; K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code.” IEEE Transactions on Software Engineering, 2002, vol. 28, no 7, pp. 654-670.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, & L. Bier, “Clone detection using abstract syntax trees”. Software Maintenance, 1998. Proceedings, International Conference, pp. 368-377. IEEE.

[5] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, & M. Bernstein, “Pattern matching for clone and concept detection”. Automated Software Engineering, 1996, vol. 3, no 1-2, pp. 77-108.

[6] N. Davey, P. Barson, S. Field, R. Frank, & D. Tansley, “The development of a software clone detector”. International Journal of Applied Software Technology, 1995.

[7] “The Scala Programming Language”, https://www.scala-lang.org/ (Consultado Diciembre 2017).

[8] “Scala vs Java - Differences and Similarities”, Javin Paul, http://javarevisited.blogspot.com.es/2013/11/scala-vs-java-differences-similarities-books.html (Consultado Diciembre 2017).

[9] “IntelliJ IDEA: The Java IDE”, https://www.jetbrains.com/idea/ (Consultado Enero 2018)

[10] “Bitbucket, the Git solution for professional teams”, https://bitbucket.org/ (Consultado Enero 2018)

[11] “GumTreeDiff/gumtree: A neat code differencing tool”, https://github.com/GumTreeDiff/gumtree (Consultado Enero 2018)

[12] “JRuby's parser customized for IDE usage”, https://github.com/jruby/jruby-parser (Consultado Febrero 2018)

[13] “ScalaFX simplifies creation of JavaFX-based user interfaces in Scala.”, https://github.com/scalafx/scalafx (Consultado Febrero 2018)

[14] “JavaFX tutorial”, https://www.tutorialspoint.com/javafx/index.htm (Consultado Febrero 2018)

[15] D. L. McCabe, “Cheating among college and university students: A North American perspective”. International Journal for Educational Integrity, 2005, vol. 1, no 1.

[16] Y. Jia, “Clone Detection Using Dependence Analysis and Lexical Analysis” Department of Computer Science, King’s College London, September 2007, http://www0.cs.ucl.ac.uk/staff/mharman/PastMScProjects2006/YueJia.pdf (Consultado Marzo 2018)

Glosario

API	Application Programming Interface (Interfaz de Programación de Aplicaciones)
AST	Abstract Syntax Tree (Árbol de Sintaxis Abstracta)
Backend	Capa de acceso a los datos en una aplicación
CFG	Control Flow Graph (Grafo de Flujo de Control)
DRY	Principio de desarrollo de software caracterizado por la reducción de las repeticiones de código (Don't Repeat Yourself)
Framework	Entorno de trabajo, que incluye tanto una manera concreta de trabajar como puede incluir un conjunto de librerías.
Frontend	Capa de interfaz
IDE	Integrated Development Environment (Entorno de Desarrollo Integrado)
JVM	Java Virtual Machine
PDG	Program Dependency Graph (Grafo de Dependencias del Programa)
POC	Proof Of Concept (Prueba de Concepto)
NP	Clase de complejidad de un problema que puede ser resuelto en tiempo polinómico por una máquina de Turing.